Formalizing Design Spaces: Implicit Invocation Mechanisms

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA

David Notkin
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA 98195 USA

Abstract

An important goal of software engineering is to exploit commonalities in system design in order to reduce the complexity of building new systems, support large-scale reuse, and provide automated assistance for system development. A significant roadblock to accomplishing this goal is that common properties of systems are poorly understood. In this paper we argue that formal specification can help solve this problem. A formal definition of a design framework can identify the common properties of a family of systems and make clear the dimensions of specialization. New designs can then be built out of old ones in a principled way, at reduced cost to designers and implementors.

To illustrate these points, we present a formalization of a system integration technique called implicit invocation. We show how many previously unrelated systems can be viewed as instances of the same underlying framework. Then we briefly indicate how the formalization allows us to reason about certain properties of those systems as well as the relationships between different systems.

1 Introduction

Software systems are rarely conceived in isolation. Instead, most systems represent a new instance of a product in some family of related systems, be they accounting systems, database products, or compilers. Each system typically shares with other applications in its family a common "framework" of behavioral and structural properties. An important goal of software engineering is to improve software productivity by exploiting these frameworks to reduce the complexity of building systems, support large-scale reuse, and provide automated assistance for system development.

Broadly speaking, we have not been very successful in doing this. With the exception of certain specialized domains (eg., spreadsheets, report generation, compiler construction), most systems are treated as an innovative development effort. Commonalities in design, if exploited at all, influence development primarily through past experience of builders who apply their knowledge in ad hoc and unstructured ways. As a result there is a proliferation of system designs and implementation mechanisms, even when the resulting products have many features in common.

There are two fundamental reasons for our failure to exploit commonalities in design. First, differences in system design often reflect important differences in system requirements. As an example, consider the diversity of tool integration mechanisms found in programming environments. Even though most modern environments share a common

goal of integrating a set of program development tools, an environment that is required to incorporate existing tools may well lead to a different design for tool integration than one in which all tools are hand-crafted for the environment.

The second reason for the proliferation of designs, however, is that the common properties of those designs are poorly understood. Consequently it is hard to take advantage of previous experiences when designing a system, even though the new system may have much in common with existing systems. (Approaches such as object-oriented design, JSP/JSD, etc., do not directly address this problem, since they give guidance for how to construct specific designs rather than for understanding or relating families of designs.)

In this paper we argue that formal specification can help solve this problem. First, a formal definition of a design framework can identify the common properties of a family of systems. This makes it possible to see how previously unrelated systems can be treated as instances of the same underlying design. Second, if carried out properly, the formalization can make clear the dimensions of specialization that can be used to turn an abstract design framework into a design for a particular application. In this way, new designs can be built out of old ones in a principled way, at reduced cost to designers and implementors.

To illustrate these points, we present a formalization of a system integration technique, called implicit invocation. We show how many previously unrelated systems can be viewed as instances of the same underlying framework. Then we briefly indicate how the formalization allows us to reason about certain properties of those systems as well as the relationships between different systems.

2 Implicit Invocation Mechanisms

Complex systems are typically composed out of many components, such as data types, objects, tools, knowledge sources, etc. A fundamental issue in the development of these systems is the choice of mechanism for integrating those components. Certainly the most common integration mechanism is explicit invocation: components interact directly by calling routines in other components. These "routines" may be interface routines of abstract data types, invocation commands of tools, methods of objects, explicit queries of databases, etc.

However, there is another mechanism — implicit invocation — that is also becoming a widespread technique for organizing systems. The idea behind implicit invocation is that actions performed by one component in a system may cause the invocation of routines in other components in the system, without the original component having explicit static references to those other components. (We will make the definitions of the terms "action" and "component" precise later.)

For example, in the Field system [Reiss 90], tools such as editors and variable monitors can register interest in a debugger's actions related to breakpoints. Then, when the debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke methods in those registered tools.[1] These methods might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools (if any) are concerned with that event, or what they will do when that event is

[1] In the remainder of this paper we will refer to the interface routines of a component as its "methods".

announced.

The widespread interest in implicit invocation mechanisms arises for at least two reasons. First, implicit invocation mechanisms make it easier to build systems that relieve users from having to explicitly invoke related components. For example, a user interface might use implicit invocation to automatically update multiple views of the same data when the user changes that data. Implicit invocation makes systems like these easier to build because complex interactions between components need not be directly encoded in the components themselves. In this way, components can be built largely independently, but still work together in supporting a user's goals. Second, implicit invocation mechanisms reduce the cost of system evolution [Sullivan & Notkin 90]. In particular, because components are loosely coupled, it is possible to integrate new components without affecting the components that implicitly invoke the new components. This makes it possible to evolve an environment more easily over time, as well as to configure an environment dynamically.

Examples of systems with implicit invocation mechanisms abound. They are used in programming environments to integrate tools [Reiss 90, Gerety 89], in database management systems to ensure consistency constraints [Hewitt 69, Balzer 86], in user interfaces to separate presentation of data from applications that manage the data [Krasner & Pope 88], and by syntax-directed editors to support incremental semantic checking [Habermann & Notkin 86, Habermann et al. 91]. They also appear in spreadsheets, blackboard systems, attribute grammar systems, constraint-based systems, as well as many other kinds of systems.

Typically each such implicit invocation mechanism has been produced as a separate and innovative design, at significant intellectual and developmental cost. As we mentioned earlier, one of the reasons for this duplication of effort is that those mechanisms have rather different goals. However, a perhaps more significant reason is that commonalities between the mechanisms have not been understood, or, until recently, even recognized [Garlan et al. 90]. In the remainder of this paper we show how formal specification can help make these relationships clear. We present an abstract model of implicit invocation, based on the notion of event announcement. Then we illustrate how specific designs are obtained by elaborating this abstract model.

3 A Formal Model for Implicit Invocation

Abstractly, an implicit invocation mechanism can be modelled as a collection of components, each of which has an interface that specifies a set of methods and a set of events. As is traditionally the case, the methods define operations that other components can explicitly invoke. The events, however, define actions that the component promises to announce to other components in the system. The implicit invocation mechanism must also provide a way to associate events with methods that are to be invoked when those events are announced.

We now formalize these ideas in three steps using the Z specification language. First, we define the state space of the basic model. Next we define an abstract run-time model that explains how events are announced and handled by an implicit invocation system. Finally, we show how the model can be used to define existing systems, additionally illustrating how it allows us to reason about relationships between some of these systems.

3.1 The Basic Model

We begin by assuming there exist sets of events, methods, and component names, which, for the time being, we will simply treat as primitive types.

$$[EVENT, METHOD, CNAME]$$

A component is modelled as an entity that has a name and an interface consisting of a set of *methods* and a set of *events*.

Component

$$\begin{array}{l} name : CNAME \\ methods : \mathbb{P}\ METHOD \\ events : \mathbb{P}\ EVENT \end{array}$$

A particular event (or method) is identified by a pair consisting of the name of a component and the event (or method) itself. In this way we can talk about the same event or method appearing in different components. We use the type abbreviations *Event* and *Method* to refer to these pairs (respectively).

$$Event == CNAME \times EVENT$$
$$Method == CNAME \times METHOD$$

For convenience we define the functions *Events* and *Methods*, which extract the set of components and methods from a collection of components.

$$\begin{array}{l} Events : \mathbb{P}\ Component \rightarrow \mathbb{P}\ Event \\ Methods : \mathbb{P}\ Component \rightarrow \mathbb{P}\ Method \\ \hline Events\ cs = \{c : cs;\ e : EVENT \mid e \in c.events \bullet (c.name, e)\} \\ Methods\ cs = \{c : cs;\ m : METHOD \mid m \in c.methods \bullet (c.name, m)\} \end{array}$$

An event system, *EventSystem*, consists of a set of components and an event manager. The event manager, *EM*, is a binary relation associating events and methods that should be invoked when that event is announced. Thus, as we will see later, when an event e is announced, all methods related to it by *EM* are invoked in the corresponding components.

EventSystem

$$\begin{array}{l} components : \mathbb{P}\ Component \\ EM : Event \leftrightarrow Method \\ \hline \forall c_1, c_2 : components \bullet (c_1.name = c_2.name) \Leftrightarrow (c_1 = c_2) \\ \mathrm{dom}\ EM \subseteq Events\ components \\ \mathrm{ran}\ EM \subseteq Methods\ components \end{array}$$

The state invariant of *EventSystem* asserts that the components in the system have unique names, and that the event manager contains only events and methods that actually exist in the system.

This characterization of *EM* is an extremely general one. In particular, this model allows the same event to be associated with many different methods, and even with many methods in the same component. It also permits some events to be associated with no methods. Further, it leaves open the issue of what components can announce events, and whether there are any restrictions on the methods that can be associated with those events

3.2 Run-Time Model.

To give meaning to *EventSystem* it is necessary to say how it behaves. There are two basic operations to consider. The first operation allows a component to "announce" an event, and the second abstractly allows the system to choose an event and then invoke the methods associated with it.

Both operations are accommodated by a run-time model for an event system that associates a set of announced events with that system. This set contains events that have been raised but not yet been handled. We will insist, however, that announced events are in fact ones that the event manager can handle.

RunTimeSystem

$$
\begin{array}{|l}
EventSystem \\
announced : \mathbb{P}\ Event \\
\hline
announced \subseteq Events\ components \\
\hline
\end{array}
$$

Announcing an event is straightforward: an component can only announce one of its own events, and the effect is simply to add the event to *announced*.

Announce

$$
\begin{array}{|l}
\Delta RunTimeSystem \\
announcer? : Component \\
event? : EVENT \\
\hline
announcer? \in components \\
event? \in announcer?.events \\
\Xi EventSystem \\
announced' = announced \cup \{(announcer?.name, event?)\} \\
\hline
\end{array}
$$

The run-time behavior of an event system is determined largely by the run-time behavior of the components—that is, by the way they handle method invocations. There are many possible models for this behavior. However, for the purpose of abstractly modelling an event system, we simply assume the existence of a run-time operation *InvokeMethods* that, given a set of methods, invokes them in some system-specific way to change the state of the components in the system.

InvokeMethods

$$
\begin{array}{|l}
\Delta RunTimeSystem \\
methods : \mathbb{P}\ Method \\
\hline
methods \subseteq Methods\ components \\
\hline
\end{array}
$$

We assume as a precondition of *InvokeMethods* that the methods to be invoked are *bone fide* methods of the *EventSystem*, but say nothing about the result of having invoked those methods.

This definition naturally leaves open many aspects of *InvokeMethods*. Any implementation of such a system would have to make numerous decisions, such as the order in which the methods are invoked, whether methods can be invoked concurrently, whether methods can change the set of components in the system, how new events are announced as a side effect of method invocation, etc.

Events are handled in two phases. First, an event is selected and removed from the set of announced events. No other aspect of the run-time system changes during this phase. Second, the selected event is "propagated" by invoking the methods associated with that event.

Select

$$\Delta RunTimeSystem$$
$$e' : Event$$

$$e' \in announced$$
$$announced' = announced \setminus \{e'\}$$
$$\theta EventSystem = \theta EventSystem'$$

We intentionally do not say how the event is chosen, but leave that decision to vary from system to system.

Propagate

$$\Delta RunTimeSystem$$
$$e : Event$$

$$\exists\, methods : \mathbb{P}\, Method \mid methods = EM(\!|\{e\}|\!) \bullet InvokeMethods$$

Finally, we put the two parts together to obtain the operation *HandleEvent*.

$$HandleEvent \mathrel{\widehat{=}} Select \fatsemi Propagate$$

In these definitions we do not place any restrictions on the computations performed by the components through explicit (method) invocation as a consequence of *HandleEvent*.

This definition of a run-time model is so abstract as to be of little direct practical use, although it is needed to make clear the context in which an event system is used. In fact, the schemas defining the model can be thought of as placeholders that would require specialization to define a given concrete system. However, rather than discuss these run-time specializations, in the remainder of the paper we focus instead on the specializations of *EventSystem* itself.

3.3 Specializations

To show how this general framework can be specialized we consider five specific systems. The first system is the basis for the user interface design of Smalltalk-80. The next two are examples associated with databases in programming environments. The last two are tool integration mechanisms for systems constructed out of existing Unix-based tools. For purposes of illustration and comparison we will focus primarily on programming environment examples, although the model applies equally well to a much broader class of systems [Garlan et al. 90]. In each case we will refine the definition of *EventSystem* in one of two ways: by elaborating the elements of *EVENT* and *METHOD*; and by constraining the way *EM* is defined.

Smalltalk-80 MVC. Our first specialization is the implicit invocation mechanism that supports the Smalltalk-80 Model-View-Controller (MVC) paradigm [Goldberg & Robson 83]. This mechanism is based on the notion that any object can register as a

"dependent" of any other object. When an object announces the "changed" event, the "update" method is implicitly invoked in each of its dependents. Thus the MVC provides a fixed, predetermined set of events (namely the "changed" event) and associated methods (namely the "update" method).[2]

To model this mechanism formally we first state that the *changed* event and *update* method are elements of types *EVENT* and *METHOD*, respectively.

$$changed : EVENT$$
$$update : METHOD$$

Next, we model dependencies between objects as a relation between components. This dependency relation then precisely determines the *EM* relation as follows: first, the events associated with each component is restricted to the set $\{changed\}$; and second, *EM* simply pairs *changed* events with the appropriate *update* methods.

*ST*80

$$EventSystem$$
$$dependents : Component \leftrightarrow Component$$

$$\mathrm{dom}\ dependents \subseteq components$$
$$\mathrm{ran}\ dependents \subseteq components$$
$$\forall c : components \bullet c.events = \{changed\}$$
$$EM = \{c_1, c_2 : components \mid (c_1, c_2) \in dependents \bullet ((c_1.name, changed), (c_2.name, update))\}$$

Note that a consequence of the invariant of *EventSystem* is that each dependent in the system must have *update* as one its methods. We could formulate this as a lemma—albeit a simple one—to be proved about such a system. In its implementation, Smalltalk-80 supports this lemma by providing a default *update* method in the Object class, at the top of the class hierarchy.

APPL/A. The next specialization is the implicit invocation mechanism of APPL/A [Sutton, Heimbigner & Osterweil 90], an Ada-based language that supports process programming. APPL/A's implicit invocation mechanism was inspired by mechanisms to support consistency in entity-relation databases [Hewitt 69, Balzer 86]. Its event mechanism has two basic restrictions. First, relations—instances of a special type constructor in APPL/A—are the only components that can announce events. Other components, such as variables, packages, and tasks, communicate only through explicit invocation. Second, a relation can only announce one of the following predefined events: insert, delete, update, find.[3] These events correspond to the primitive operations that can be performed on relations.

To model this formally we declare the events specific to this mechanism.

$$insert\_e, delete\_e, update\_e, find\_e : EVENT$$

[2]Actually, there are three versions of the *changed* event: one with no parameters, one with one parameter, and one with two parameters. There are also three corresponding versions of the *update* method. Also, it should be pointed out that although Smalltalk-80 is flexible enough to program alternative implementations of an implicit invocation mechanism, the MVC paradigm does not take advantage of this flexibility.

[3]Actually, each of these four event types has two flavors: accept and complete.

Next, we identify a distinguished set of components, called *relations*. These are the only components that can announce events and hence all other components in the system have an empty set of events. These restrictions are summarized in the schema *APPL_A*.

APPL_A

$$
\begin{array}{l}
\mathit{EventSystem} \\
\mathit{relations} : \mathbb{P}\ \mathit{Component} \\
\hline
\mathit{relations} \subseteq \mathit{components} \\
\forall r : \mathit{relations} \bullet r.\mathit{events} = \{\mathit{insert\_e}, \mathit{delete\_e}, \mathit{update\_e}, \mathit{find\_e}\} \\
\forall c : \mathit{components} \setminus \mathit{relations} \bullet c.\mathit{events} = \emptyset
\end{array}
$$

Gandalf. The third mechanism is the "daemon" invocation mechanism of the Gandalf System [Habermann & Notkin 86]. Gandalf uses implicit invocation to provide (among other things) incremental, static semantic checking for programs. The user creates a program by incrementally building an abstract syntax tree. As nodes are added to the tree, daemons associated with those nodes are activated to do type checking, provide incremental code generation, etc.

Gandalf is similar to APPL/A insofar as it defines a fixed set of events. (Likewise, these events correspond to the primitive operations that can be performed on nodes in the abstract syntax tree.) For the purposes of this exposition, we adopt the same set. Gandalf also imposes some additional structure. First, as we have said, Gandalf has two kinds of components: abstract syntax trees (ASTs) and daemons. Second, only the nodes in an AST can announce events. Third, each node in an AST is paired with at most one daemon, which is responsible for handling all events announced by its node. Thus each daemon has a set of methods corresponding precisely to the events that a node can announce.

To model the Gandalf implicit invocation mechanism we define the names of methods supported by daemons (*insert*, *delete*, ...) , and then restrict the event mechanism as indicated.[4]

$$
\begin{array}{l}
\mathit{Node}, \mathit{Daemon} : \mathbb{P}\ \mathit{Component} \\
\mathit{insert\_m}, \mathit{delete\_m}, \mathit{update\_m}, \mathit{find\_m} : \mathit{METHOD}
\end{array}
$$

[4] In these definitions we assume there exists a generic type constructor *TREE* and a function *nodes* that yields the set of nodes in a tree. These are easily added to the mathematical toolkit, in the style of the Z Reference Manual [Spivey 89].

Gandalf

$$
\begin{array}{l}
\textit{EventSystem} \\
\textit{AST} : \textit{TREE}[\textit{Node}] \\
\textit{daemons} : \mathbb{P}\ \textit{Daemon} \\
\textit{ND} : \textit{Node} \nrightarrow \textit{Daemon} \\
\hline
\mathrm{dom}\ \textit{ND} \subseteq \textit{nodes}\ \textit{AST} \land \mathrm{ran}\ \textit{ND} = \textit{daemons} \\
\textit{daemons} \cap \textit{AST} = \emptyset \\
\forall\, n : \textit{AST} \bullet n.\textit{events} = \{\textit{insert\_e}, \textit{delete\_e}, \textit{update\_e}, \textit{find\_e}\} \\
\forall\, d : \textit{daemons} \bullet d.\textit{methods} = \{\textit{insert\_m}, \textit{delete\_m}, \textit{update\_m}, \textit{find\_m}\} \\
\textit{EM} = \{n : \mathrm{dom}\ \textit{ND} \bullet ((n.\textit{name}, \textit{insert\_e}), ((\textit{ND}\ n).\textit{name}, \textit{insert\_m}))\} \cup \\
\qquad \{n : \mathrm{dom}\ \textit{ND} \bullet ((n.\textit{name}, \textit{delete\_e}), ((\textit{ND}\ n).\textit{name}, \textit{delete\_m}))\} \cup \\
\qquad \ldots
\end{array}
$$

Field. The next system is the tool integration mechanism developed by the Field System [Reiss 90]. Field was designed to make it relatively easy to incorporate existing Unix tools into a programming environment. In a Field environment tools communicate by "broadcasting" interesting events. Other tools can register patterns that indicate which events should be routed to them and which methods should be called when an event matches that pattern. When an event is announced, a pattern matcher checks all registered patterns, invoking the associated method whenever a pattern is matched. For example, if a program editor announces when it has finished editing a module, a compiler might register for such announcements and automatically recompile the edited module.

To describe this behavior in terms of our basic model, we first define a new type of basic entity, *PATTERN*.

$$[\textit{PATTERN}]$$

Next we associate a pattern matcher (*match*) with *EventSystem*, and a *register* relation that, for each component, associates patterns with methods of that component. The register relation then uniquely determines *EM*.

Field

$$
\begin{array}{l}
\textit{EventSystem} \\
\textit{match} : \textit{EVENT} \leftrightarrow \textit{PATTERN} \\
\textit{register} : \textit{Component} \leftrightarrow (\textit{PATTERN} \times \textit{METHOD}) \\
\hline
\mathrm{dom}\ \textit{register} \subseteq \textit{components} \\
((c_1.\textit{name}, e), (c_2.\textit{name}, m)) \in \textit{EM} \Leftrightarrow \\
\qquad (\exists\, \textit{pat} : \textit{PATTERN} \bullet (c_2, (\textit{pat}, m)) \in \textit{register} \land (e, \textit{pat}) \in \textit{match})
\end{array}
$$

The invariant guarantees that the Event/Method pairs in *EM* are those for which some registered pattern matches the event associated with the method.

Forest. Forest adapts the Field implicit invocation mechanism to allow each component to define a "policy" for deciding (dynamically) which methods to invoke when a pattern is matched [Garlan & Ilias 90]. Thus instead of always invoking the same

method, a component's policy determines what method should be invoked. The policy evaluation may depend on some state variables maintained by the event system.

For example, incremental, implicit recompilation of edited modules—as illustrated for Field— may be appropriate at certain times but not others. Using Forest, it is possible to define a policy to control when the compiler should be invoked. For instance, such a policy might be defined in terms of a state variable "EditingManyModules": incremental recompilation will take place only if the value of this variable is false.

We model Forest by considering *STATE* to be a new primitive type, and by treating a *POLICY* as a function that returns a method when applied to a given *STATE*.

$$[STATE]$$
$$POLICY == STATE \rightarrow METHOD$$

Like Field, Forest has *register* and *match* relations, which together uniquely determine *EM* for the system.

Forest

$$\begin{array}{l} EventSystem \\ match : EVENT \leftrightarrow PATTERN \\ register : Component \leftrightarrow (PATTERN \times POLICY) \\ policy\_state : STATE \\ \hline \mathrm{dom}\ register \subseteq components \\ ((c_1.name, e), (c_2.name, m)) \in EM \Leftrightarrow \\ \quad (\exists\, pat : PATTERN;\ policy : POLICY \bullet \\ \qquad (c_2, (pat, policy)) \in register \wedge \\ \qquad (e, pat) \in match \wedge \\ \qquad policy(policy\_state) = m) \end{array}$$

As with Field, the invariant of Forest guarantees that the Event/Method pairs in *EM* are those for which both a pattern and a policy combine to map the event to the method.

3.4 Using the Model

The value of the implicit invocation model is its ability to provide insight into the space of designs that adopt implicit invocation mechanisms. At the very least, the formal abstractions greatly simplify comparisons between existing systems, and make clear how each of them can be viewed as a variation on a common underlying design. Designs that are typically realized as thousands of lines of code, can be examined and compared with relative ease in several pages of formal specification.

For example, similarities between Gandalf and APPL/A become apparent, even though the two systems were implemented completely independently, and with quite different goals. In particular, both generate events in response to operations on certain types of components—relations, in the case of APPL/A, and AST nodes, in the case of Gandalf. Both restrict the components in the system that can handle events, and both restrict the vocabulary of event announcements. Finally, both are similar to Smalltalk-80, which further restricts announcers and receivers of events.

However, the usefulness of a formal model extends beyond qualitative, high-level comparisons. Indeed, there are at least three important ways in which the model allows us to reason about different implicit invocation systems.

First, it makes explicit the restrictions that each system imposes on the general model. This in turn allows us to explain some of the limitations of existing systems, and to predict properties of new designs. For example, consider APPL/A's restriction that allows only relations to announce events. A consequence is that the programmer either must use standard, explicit invocation mechanisms (eg., procedure call) to connect components that are not relations, or else must model those components as relations. In the former case, the benefits of implicit invocation mechanisms—such as easier evolution—may not apply. In the latter case, system components are forced to be treated as relations, even when they have other, more natural representations.

As another example, consider Gandalf's restriction of *EM*. It is not hard to show that *EM* is a (partial) function. That is,

$$Gandalf \vdash EM \in Event \nrightarrow Method$$

Thus in the Gandalf System only a single component (viz., a daemon) can be associated with an event. As a consequence each implicitly invoked method is itself responsible for *explicitly* invoking methods of any other component that might be "interested" in the announced event. This restriction is a serious shortcoming: since these additional relationships must be encoded in the implicitly invoked daemons, these daemons can become quite complex. Moreover, many of the benefits of loose coupling of system components is lost, making it difficult, for example, to evolve the system. Our practical experiences with Gandalf environments had given us an intuitive understanding of this general issue, but it was the formalization that clarified the specific problem.

Second, the approach allows us to formally compare systems. For example, it is possible to precisely characterize the relationship between Field and Forest. The implicit invocation mechanisms of a Field programming environment can be viewed as a special case of a Forest programming environment in which each policy always returns the same method, regardless of the *policy_state* of the system. More formally, we can state the following lemma:

$$\begin{aligned} &f : Forest;\ g : Field \vdash \\ &\quad f.components = g.components \wedge f.match = g.match \wedge \\ &\quad (\forall\ (c,(pat,m)) \in g.register \bullet \exists\ policy \mid \\ &\qquad (c,(pat,policy)) \in f.register \wedge policy(f.policystate) = m) \\ &\quad \Rightarrow f.EM = g.EM \end{aligned}$$

Third, the approach encourages the formal statement of properties that can be compared against the various specializations of the model. For instance, one such property is whether there are potential circularities in a system. In this case a circularity is a chain of implicit invocations that starts at one component and returns to that component. While the existence of circularities will, in general, depend on the specific *components* and *EM* relation in a particular *EventSystem*, for a system such as a Gandalf environment, it is possible to argue that no circularities can exist with respect to implicit invocation. (Nothing in the specification prohibits an implicitly invoked method from *explicitly* interacting with a node, which in turn may trigger the implicitly invoked method again, and so on.) More formally, given the following definition of *Circular*:

Circular

EventSystem

$$\exists\, implicitly\_invokes : components \leftrightarrow components \mid$$
$$(c_1, c_2) \in implicitly\_invokes \Leftrightarrow (\exists\, m : METHOD;\ e : EVENT \bullet$$
$$((c_1.name, e), (c_2.name, m)) \in EM) \bullet$$
$$\exists\, c : components \bullet (c, c) \in implicitly\_invokes^+$$

we can show:

$$Gandalf \vdash \neg\ Circular$$

4 Discussion and Conclusions

As we have illustrated, formal specification can provide considerable insight into a design space. This application of formal methods, however, is not a traditional one. It is therefore appropriate to reflect on this use of formalization.

Traditionally applications of formal methods have focused on the problem of developing specifications for individual systems. Moreover, typically, the primary issue is one of correctness: is a given implementation correct with respect to the given specification. In contrast, the use of formal methods described in this paper (and elsewhere [Garlan & Delisle 90, Delisle & Garlan 90a]) is to extract the abstract the properties of a *family* of systems. Abstraction allows us to concentrate on the common design decisions, ignoring specific details of particular systems, so that different systems can be compared along certain dimensions. "Refinement" is then used to show how specific systems specialize the shared abstractions. However, in this case refinement is being used to elaborate an abstract design, rather than provide an implementation of a specification. The question that one asks of a refinement is then no longer "Is it correct?", but "What are its properties?", and "How do these properties compare with those obtained through other such refinements?" Furthermore, it becomes reasonable to talk about the *dimensions of refinement* that can be used to specialize an abstract design framework. In the examples above, we looked at refinements of *EVENTS*, *COMPONENTS*, and the associated event manager. This allowed us to compare disparate systems such as Gandalf, APPL/A, and Smalltalk in terms of their refinement along these specific dimensions.

A consequence of this different emphasis on abstraction and refinement, is that the quality of a formal specification is no longer simply a question of completeness and consistency. Instead, formal specification of a design space must be judged on its ability to reveal the important properties of a family of systems. To accomplish this a formal specification must balance the simplicity of an abstract model against the need to expose key properties. This can be a difficult balance to attain. In the case of implicit invocation mechanisms we chose to abstract the way in which events are announced. This simplifies the description, but makes it difficult, for example, to describe different schemes for passing parameters through an announced event. Similarly, we chose not to model the mechanism for explicit method invocation. Again that decision leads to simplicity, but also makes it difficult for us to talk about certain hybrid implicit invocation mechanisms that, for example, announce events automatically whenever certain methods are invoked.

A second consequence is that certain properties of a formal notation become crucial in carrying out the formalization of a design space. In particular, we relied heavily on the

use of Z's schema calculus to first define a "kernel" design (*EventSystem*) and then later elaborate that design. In this respect, our use of Z is similar to Flinn and Sørensen's in their CAVIAR case study [Flinn & Ib. Sørensen]. Like our approach, they show how a specific system specification can be obtained by specializing (or instantiating) one or more reusable, abstract formal designs. The primary difference in approach, however, is that here we use the technique to support comparison of different systems. Additionally, the CAVIAR study makes heavy use of Z generic schemas to parameterize the basic building blocks. In our case study, however, we have found generics to be less useful, since the dimensions of specialization are not easily defined simply by "plugging in" the right type. This may very well point out the need for more powerful support for characterizing the parameterization of such frameworks, perhaps in the style of algebraic approaches [Goguen 86].

Acknowledgements

We would like to thank Kevin Sullivan and Robert Allen for their invaluable contributions to both the content and form of this paper. Kevin provided much of the initial insight into the underlying formal simplicity of implicit invocation mechanisms. Robert spotted numerous inconsistencies and deficiencies in earlier versions of the Z model.

References

[Balzer 86] R.M. Balzer. Living in the Next Generation Operating System. *Proceedings of the Fourth World Computer Conference.* (September, 1986).

[Delisle & Garlan 90a] N. Delisle and D. Garlan. Applying Formal Specification to Industrial Problems: A Specification of an Oscilloscope. *IEEE Software* (September 1990).

[Flinn & Ib. Sørensen] Bill Flinn and Ib. Sørensen. CAVIAR: A Case Study in Specification. In *Specification Case Studies*, ed. Ian Hayes, Prentice Hall International (1987).

[Garlan & Delisle 90] David Garlan and Norman Delisle. Formal Specifications as Reusable Frameworks. *Proceedings of the International Symposium: VDM'90 - VDM and Z.* Kiel, Germany (April 1990), Springer-Verlag, LNCS 428.

[Garlan & Ilias 90] D. Garlan and E. Ilias. Low-cost, Adaptable Tool Integration Policies for Integrated Environments. *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 1–10 (December 1990).

[Garlan et al. 90] David Garlan, Gail E. Kaiser, and David Notkin. On the Criteria to be Used in Composing Tools into Systems. Technical Report CUCS-034-90, Department of Computer Science, Columbia University (July 1990). To appear, *IEEE Computer.*

[Gerety 89] Colin Gerety. HP SoftBench: A New Generation of Software Development Tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado (November 1989).

[Goguen 86] J.A. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer* (February 1986).

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley (1983).

[Habermann et al. 91] A.N. Habermann, D. Garlan, and D. Notkin. Generation of Integrated Task-Specific Programming Environments. In *CMU Computer Science: 25th Anniversary Commemorative Symposium.* Addison-Wesley (January 1991).

[Habermann & Notkin 86] A.N. Habermann and D. Notkin. Gandalf Software Development Environments. *IEEE Transactions on Software Engineering SE-12*,12 (December 1986), pp. 1117–1127.

[Hewitt 69] Carl Hewitt. PLANNER: A Language for Proving Theorems in Robots. *Proceedings of the First International Joint Conference in Artificial Intelligence.*, Washington DC (1969).

[Krasner & Pope 88] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming 1*,3 (August/September 1988), pp. 26–49.

[Reiss 90] S.P. Reiss. Connecting Tools using Message Passing in the Field Environment. *IEEE Software* 7,4 (July 1990), pp. 57–66.

[Spivey 89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International (1989).

[Sullivan & Notkin 90] K. Sullivan and D. Notkin. Reconciling Environment Integration and Component Independence. *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 22-33 (December 1990).

[Sutton, Heimbigner & Osterweil 90] S.M. Sutton, Jr., D. Heimbigner, & L.J. Osterweil. Language Constructs for Managing Change in Process-Centered Environments. *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 206–217 (December 1990).